

# ***PADRÕES DE DESENVOLVIMENTO***

## ***DESAFIO***

Desenvolver uma classe que garanta que, dela, só exista uma única instância em execução.

# ***CURIOSIDADE***

"Grandes sistemas de software nunca são concluídos, eles simplesmente continuam evoluindo".



***O QUE FAZER?***



## ***DE MANEIRA GERAL...***

"Cada padrão descreve um **problema que ocorre repetidas vezes** em nosso ambiente, e então **descreve o núcleo da solução** para aquele problema, de tal maneira que pode-se usar essa solução milhões de vezes sem nunca fazê-la da mesma forma duas vezes"

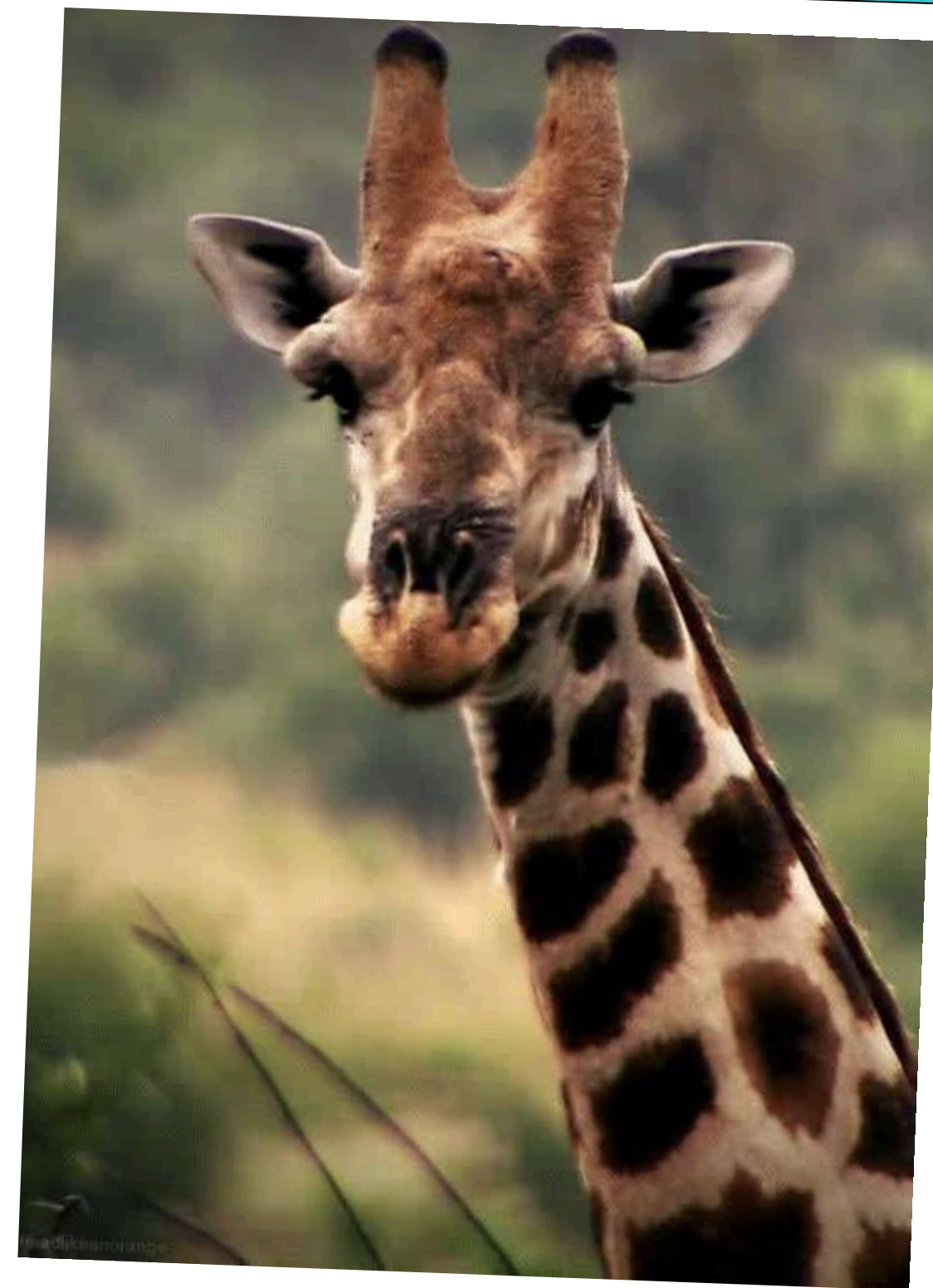
Christopher Alexander, sobre padrões em Arquitetura



## ***NA COMPUTAÇÃO***

Os padrões são soluções típicas para problemas comuns de projetos orientados a objetos, que com simples personalizações resolvemos problemas que podem ser complexos.

***O QUE MAIS TEMOS  
DE BENEFÍCIOS?***



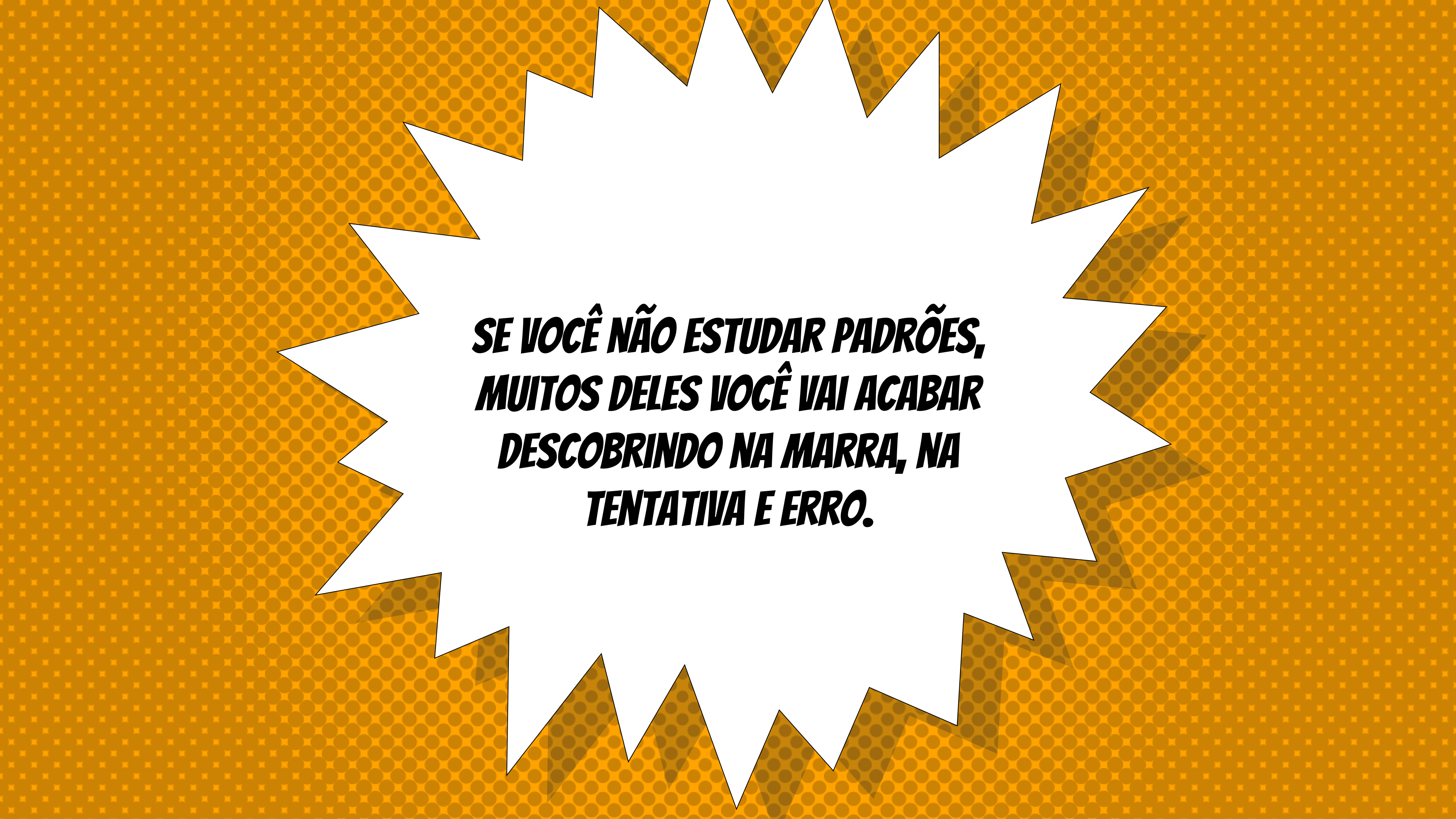
# ***BENEFÍCIOS DOS PADRÕES DE DESENVOLVIMENTO (1/2)***

- **Aprender com a experiência dos outros**
  - Não é necessário reinventar a roda
- **Aprender boas práticas de programação**
  - Padrões usam de forma eficiente herança, composição, agregação, modularidade, polimorfismo e abstração para construir software reutilizável e eficiente

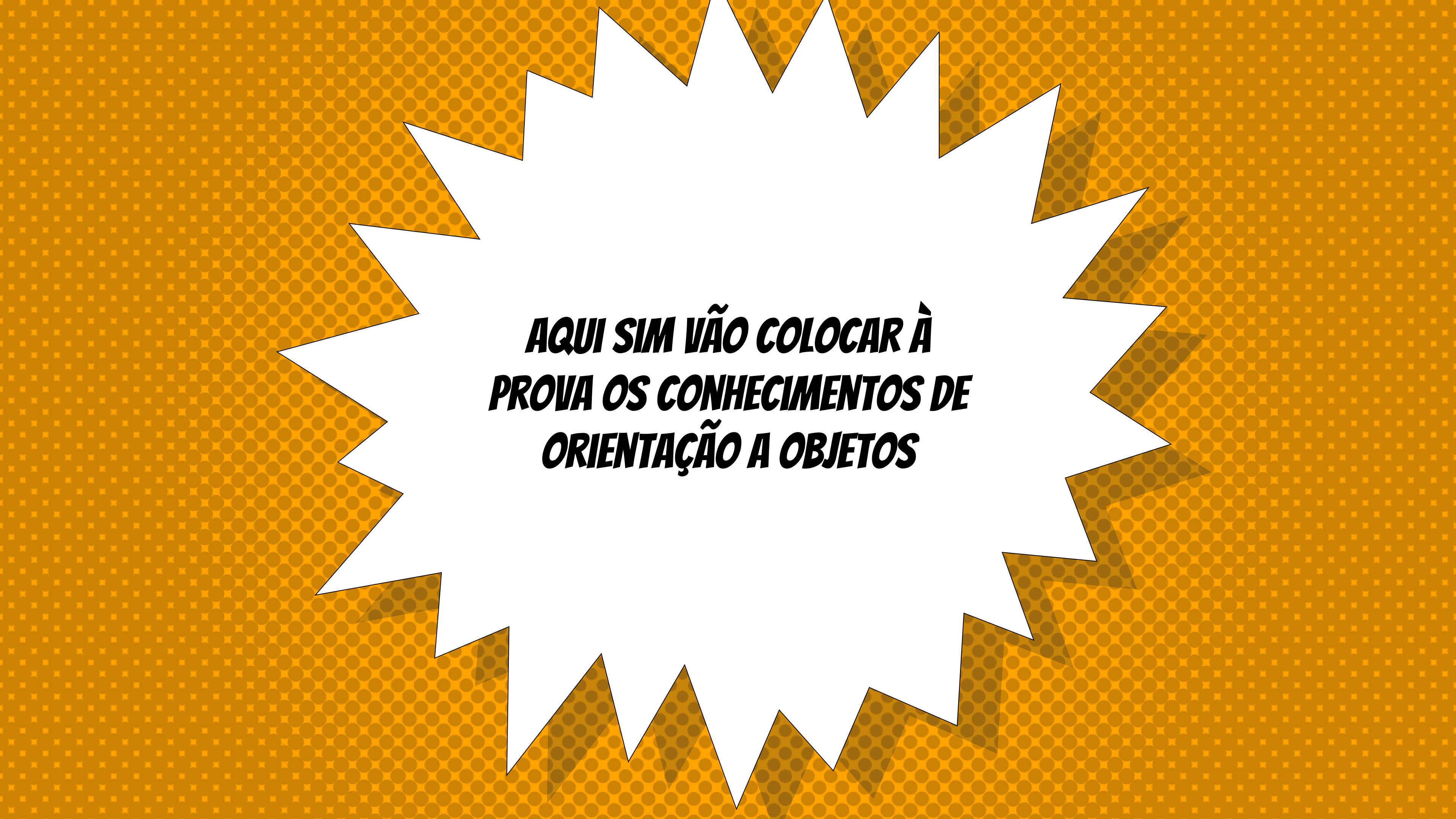


## ***BENEFÍCIOS DOS PADRÕES DE DESENVOLVIMENTO (2/2)***

- **Melhora a comunicação/compreensão**
  - Ajuda a entender o papel de classes do sistema
- **Melhora a documentação**
  - Vocabulário comum

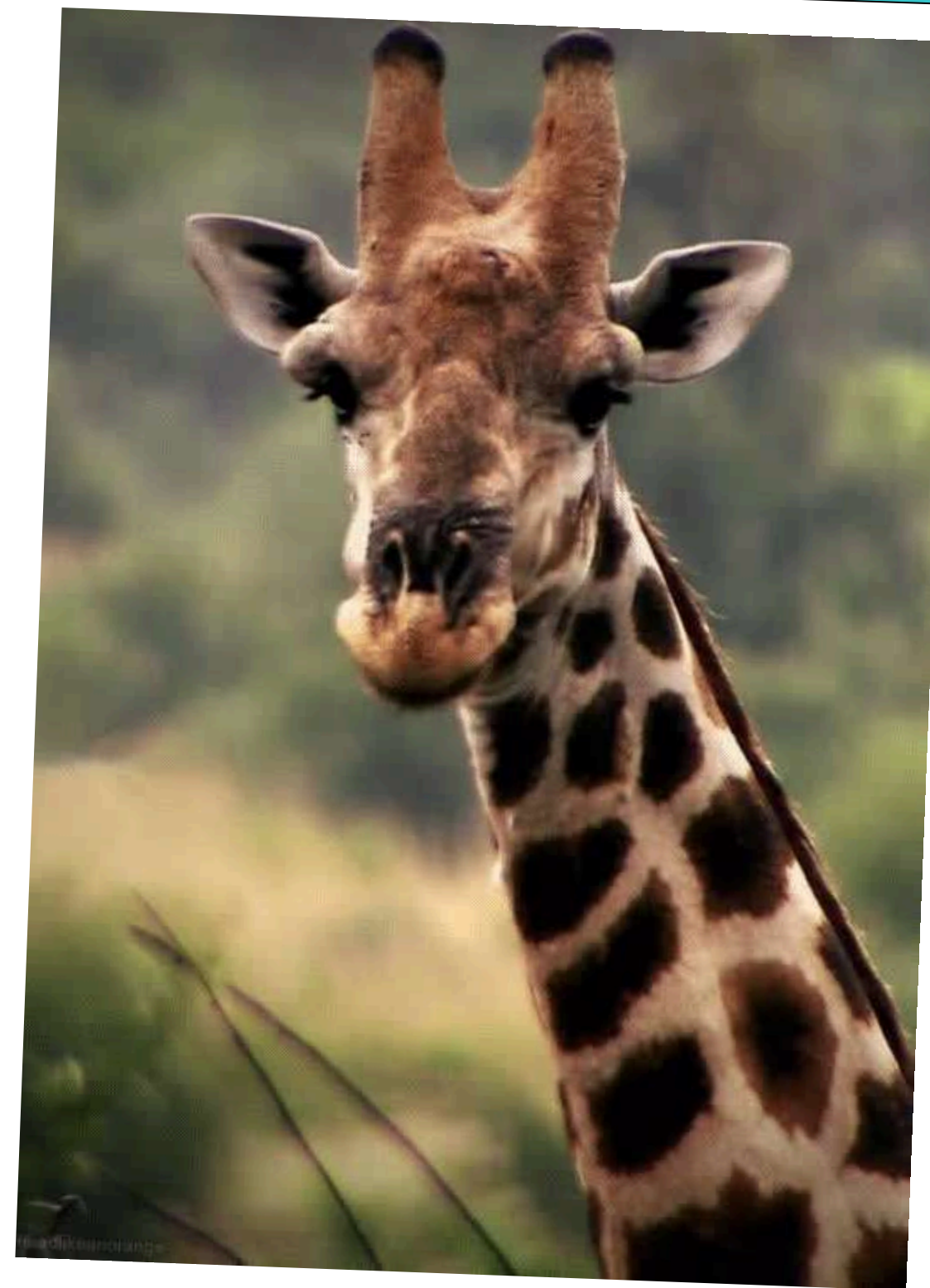


***SE VOCÊ NÃO ESTUDAR PADRÕES,  
MUITOS DELES VOCÊ VAI ACABAR  
DESCOBRINDO NA MARRA, NA  
TENTATIVA E ERRO.***



***AQUI SIM VÃO COLOCAR À  
PROVA OS CONHECIMENTOS DE  
ORIENTAÇÃO A OBJETOS***

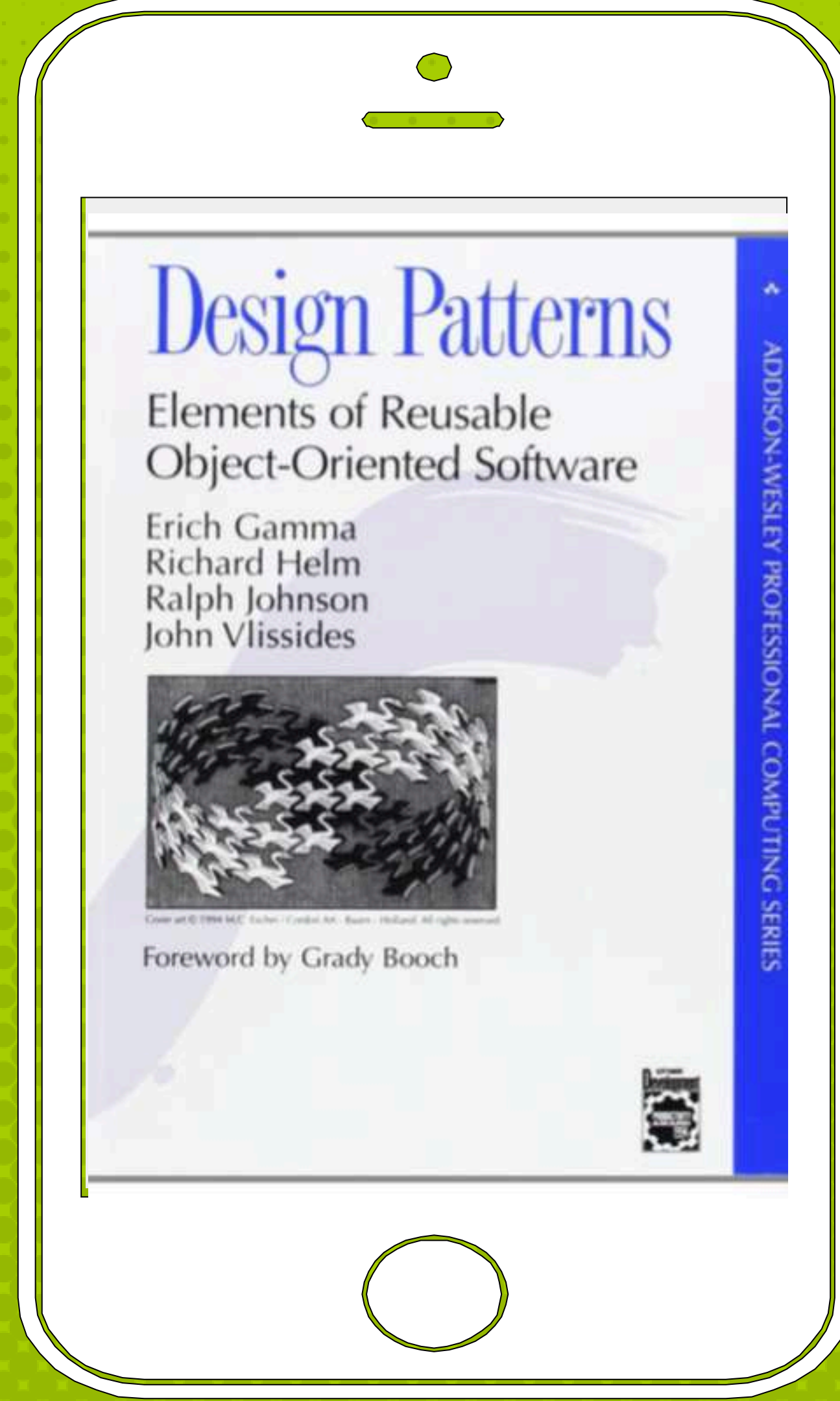
***DE ONDE VEIO ISSO?***





# COMEÇO

- Livro escrito por 4 autores, em 1994, conhecidos como "The Gang Of Four"
- O livro descreve **23 padrões**, mostrando soluções genéricas para os problemas mais comuns de desenvolvimento
- O sucesso veio da experiência dos autores, o que fez com que o livro se tornasse um **clássico da literatura OO**.



***PADRÕES GOF***

# ***OS PADRÕES GOF***

Estes estão divididos em 3 grupos

**Padrões criacionais:** Auxiliam na criação de objetos com maior flexibilidade e reutilização.

**Padrões estruturais:** Explicam como montar objetos e classes em estruturas complexas sem perder a eficiência e flexibilidade.

**Padrões comportamentais:** Cuidam da comunicação eficiente e responsabilidades entre objetos.

# OS PADRÕES GOF

Os padrões do GoF possuem **altíssima influência** sobre as bibliotecas das linguagens de programação, como exemplo Java e .NET, que possuem muitos desses padrões implementados, implícita ou explicitamente.

```
1  class Singleton {  
1  import java.util.Observable;  
2  import java.util.Observer;  
3  
4  public class RevistaInformatica extends Observable {  
5  
6      private int edicao;  
7  
8      public void setNovaEdicao(int novaEdicao) {  
9          this.edicao = novaEdicao;  
10  
11          setChanged();  
12          notifyObservers();  
13      }  
14  
15      // ...  
16  }
```



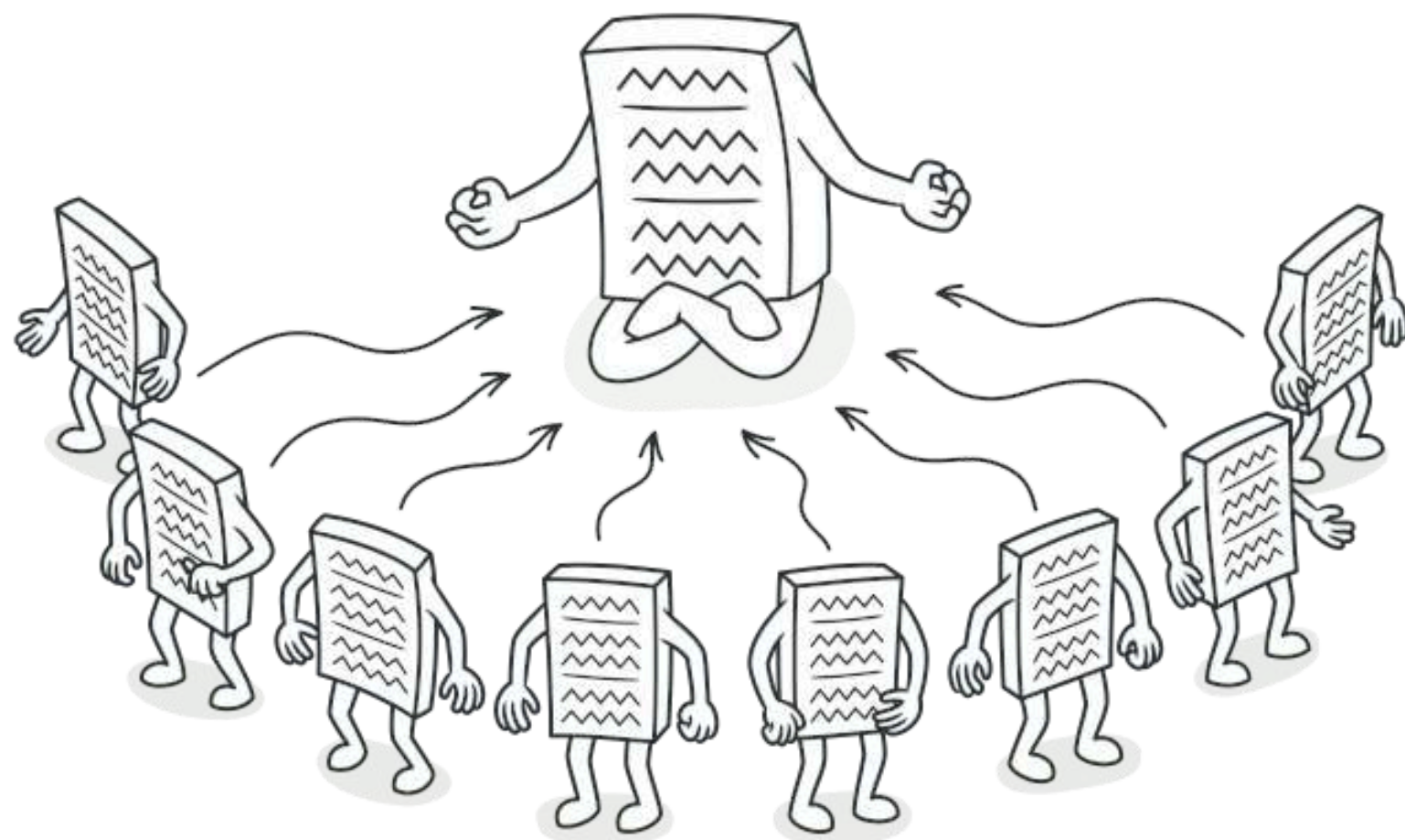
		Propósito		
		Criação	Estrutura	Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Builder Abstract Factory Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

***UM PADRÃO***

***SINGLETON***

# PROPÓSITO

Padrão do tipo **criacional** que garante que uma classe tenha **apenas uma instância** que é acessada globalmente dentro do projeto.



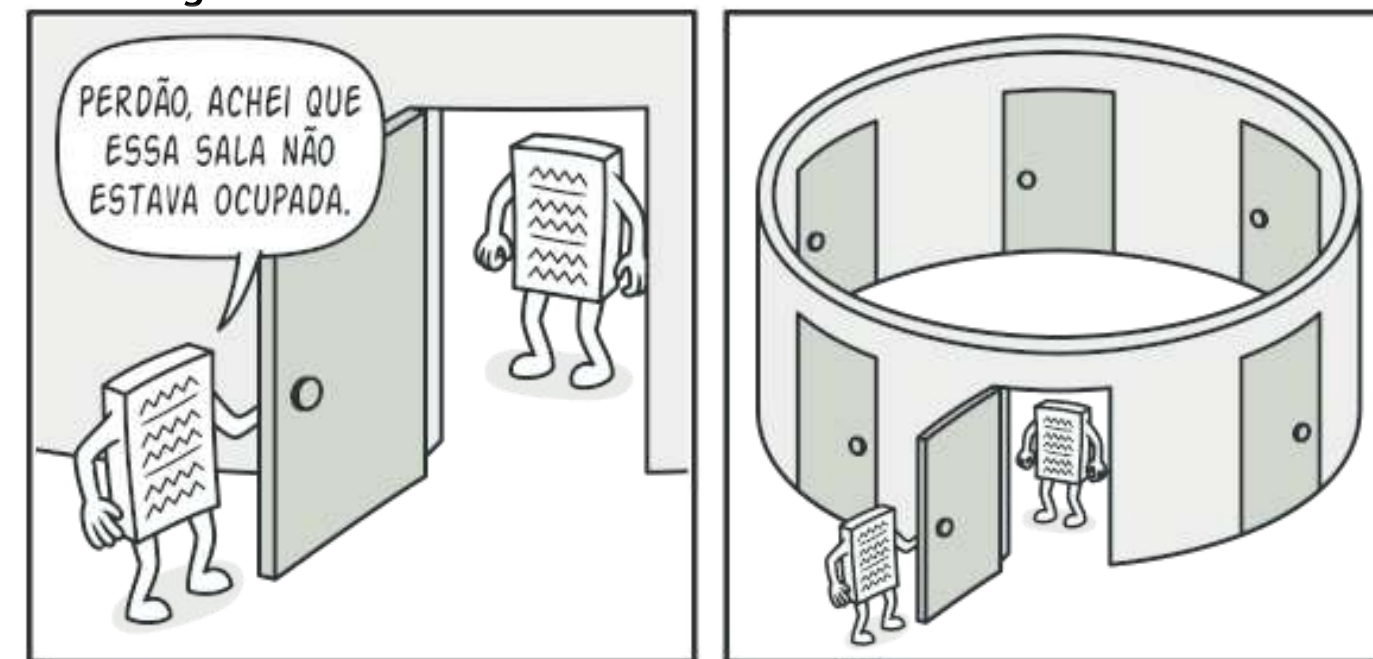
<https://refactoring.guru/pt-br/design-patterns/singleton>

# O QUE RESOLVE

Controlar o acesso a recursos **compartilhados**.

- **Garantir que uma classe tenha apenas uma instância**

Para que isso funcione, não podemos ter um construtor convencional na aplicação, pois este **sempre** cria um novo objeto.



<https://refactoring.guru/pt-br/design-patterns/singleton>



# ***O QUE RESOLVE***

O singleton permite que o objeto seja acessado de qualquer lugar do programa

- **Fornecer um ponto de acesso global para aquela instância**

**Curiosidade:** O Singleton é considerado por muitos um anti-padrão, por quebrar o princípio da responsabilidade única.

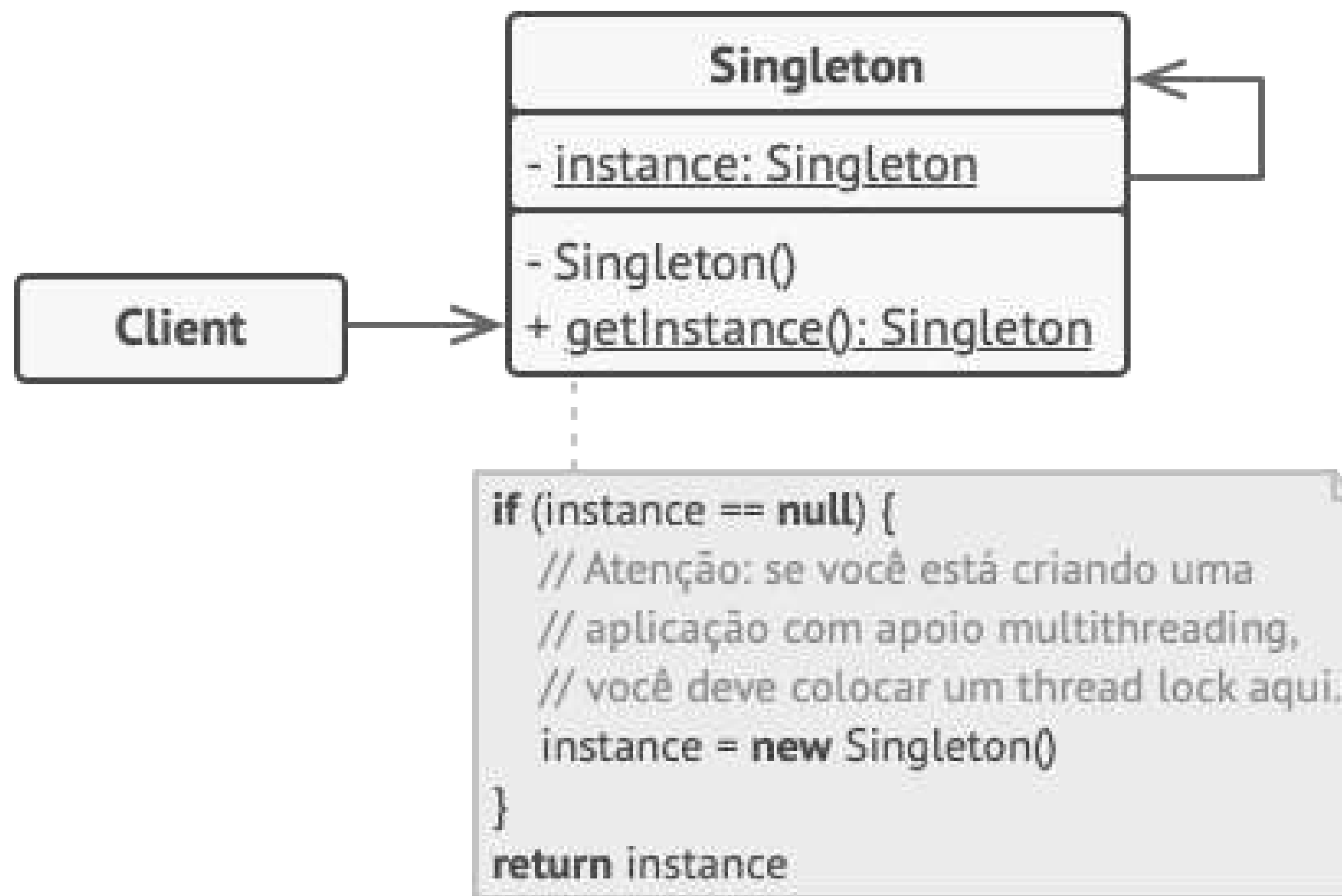
<https://refactoring.guru/pt-br/design-patterns/singleton>

# ***OBRIGATORIEDADE***

Toda implementação de Singleton deve seguir essas duas regras:

1. Ter um **construtor privado**, para que objetos não possam ser criados
2. Ter um método **estático** de criação que fará o papel do construtor. Este chama o construtor privado na primeira execução e guarda o seu objeto para utilizações futuras.

# ESTRUTURA



<https://refactoring.guru/pt-br/design-patterns/singleton>

# ***RECEITA/COMO IMPLEMENTAR***

1. Criar campo privado estático na classe para armazenamento da instância do singleton.
2. Criar método estático público para obter a instância do singleton
3. Implementar a "lazy load" no método estático. O objeto só deve ser criado na primeira chamada do método.
4. Crie um construtor privado, o método estático é quem irá chamar o construtor
5. No código cliente, substitua todas as chamadas para o construtor do singleton para as chamadas do método estático.





# ANÁLISE

## PRÓS

- Certeza de única instância da classe
- Ponto de acesso global a uma instância de classe
- A inicialização do objeto é realizada uma única vez

## CONTRAS

- Viola o princípio da responsabilidade única
- Precisa de tratamento especial em ambientes multithread
- Dificuldade na realização de testes automatizados



```
1 function clientCode() {
2   const s1 = Singleton.getInstance();
3   const s2 = Singleton.getInstance();
4
5   if (s1 === s2) {
6     console.log("Singleton funcionou, ambas variáveis possuem a mesma instância.");
7   } else {
8     console.log("Singleton falhou, as variáveis possuem instâncias diferentes");
9   }
10 }
11
12 clientCode();
```

***EM CÓDIGO***

```
1 class Singleton {
2   private static instance: Singleton;
3
4   private constructor() { }
5
6   public static getInstance(): Singleton {
7     if (!Singleton.instance) {
8       Singleton.instance = new Singleton();
9     }
10
11     return Singleton.instance;
12   }
13
14   public someBusinessLogic() {
15     // ...
16   }
17 }
```

***UM CASE DE  
OUTRO PADRÃO***

## ***UM PROBLEMA:***

### **Cálculo de estacionamento, considerando:**

Veículo de **passageio R\$ 2,00** por hora, se for mais de 12 horas, será cobrado uma **diária de R\$ 24,00**, se ficar mais do que 10 dias será cobrado uma **mensalidade de R\$ 240,00**



## UMA SOLUÇÃO:

```
package br.com.senacrs;
import static br.com.senacrs.Parametro.*;

public class ContaEstacionamento {

    private Veiculo veiculo;
    private long inicio, fim;

    public double valorConta() {
        long atual = (fim==0) ? System.currentTimeMillis() : fim;
        long periodo = inicio - atual;
        if (veiculo instanceof Passeio) {
            if (periodo < 12 * HORA) {
                return 2.0 * Math.ceil(periodo / HORA);
            } else if (periodo > 12 * HORA && periodo < 10 * DIA) {
                return 24.0 * Math.ceil(periodo / DIA);
            } else {
                return 240.0 * Math.ceil(periodo / MES);
            }
        } else if (veiculo instanceof Carga) {
            // outras regras para veículos de carga
        }
        // outras regras para outros tipos de veículo
        return Long.MAX_VALUE;
    }
}
```

# PROBLEMA:

Confuso  
Escalável (SQN)

```
package br.com.senacrs;
import static br.com.senacrs.Parametro.*;

public class ContaEstacionamento {

    private Veiculo veiculo;
    private long inicio, fim;

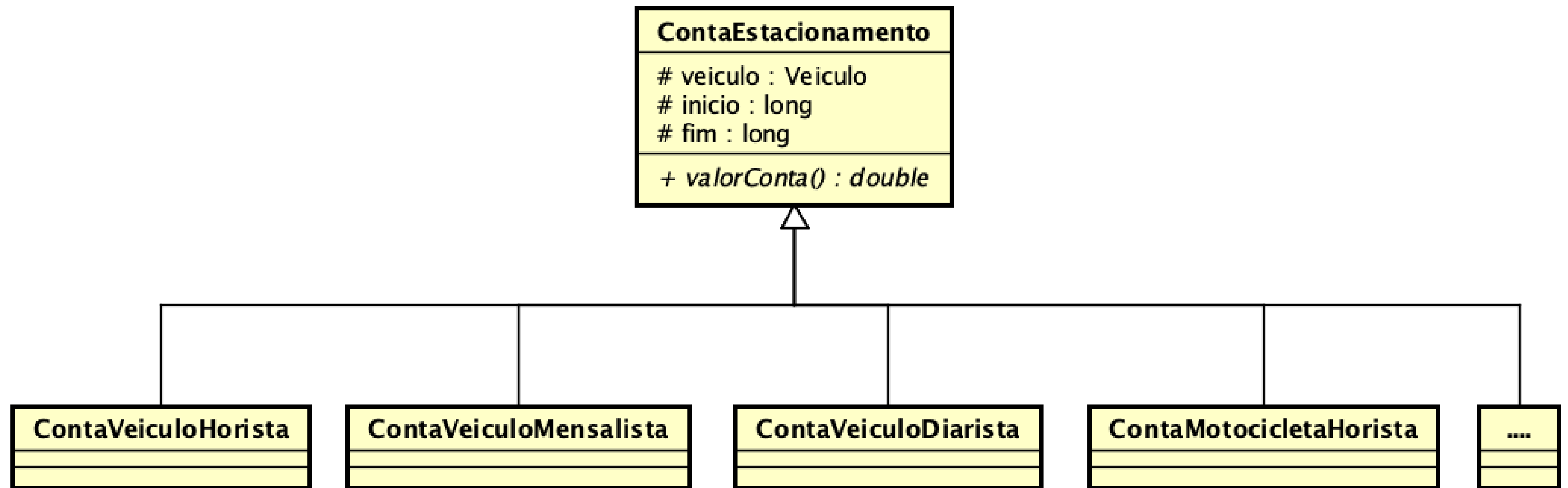
    public double valorConta() {
        long atual = (fim==0) ? System.currentTimeMillis() : fim;
        long periodo = inicio - atual;
        if (veiculo instanceof Passeio) {
            if (periodo < 12 * HORA) {
                return 2.0 * Math.ceil(periodo / HORA);
            } else if (periodo > 12 * HORA && periodo < 10 * DIA) {
                return 24.0 * Math.ceil(periodo / DIA);
            } else {
                return 240.0 * Math.ceil(periodo / MES);
            }
        } else if (veiculo instanceof Carga) {
            // outras regras para veículos de carga
        }
        // outras regras para outros tipos de veículo
        return Long.MAX_VALUE;
    }
}
```

## ***USAR HERANÇA!***

Especializar o cálculo das contas do estacionamento em subclasses de tipo de locação!



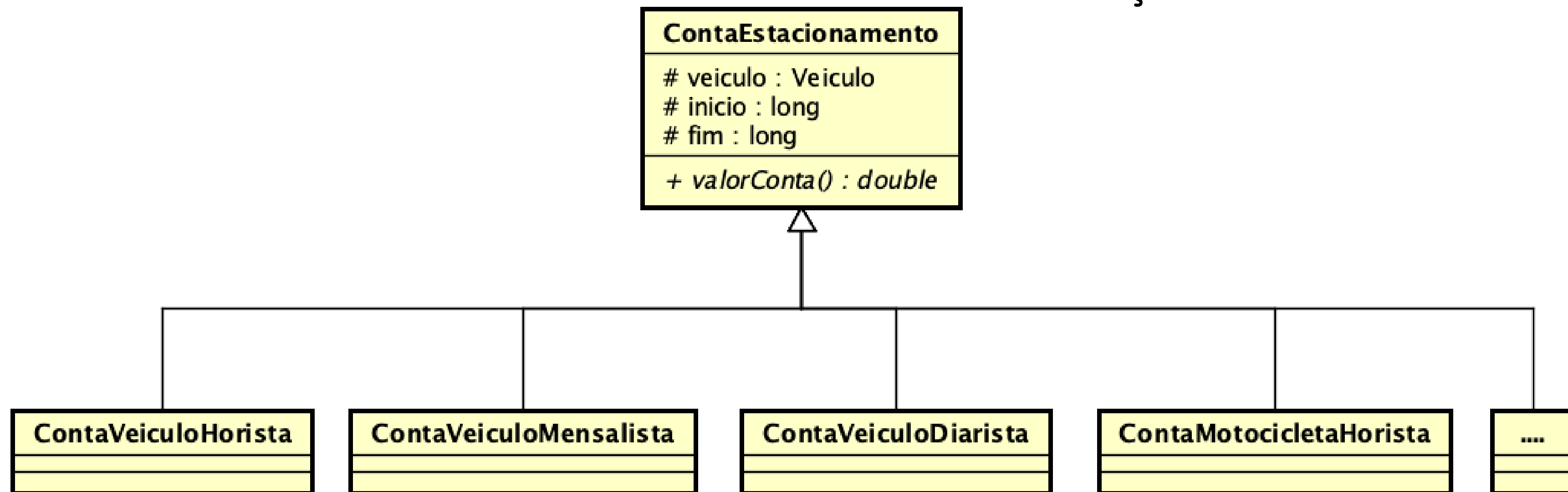
# DIAGRAMA DE CLASSES DA SOLUÇÃO





# PROBLEMAS!

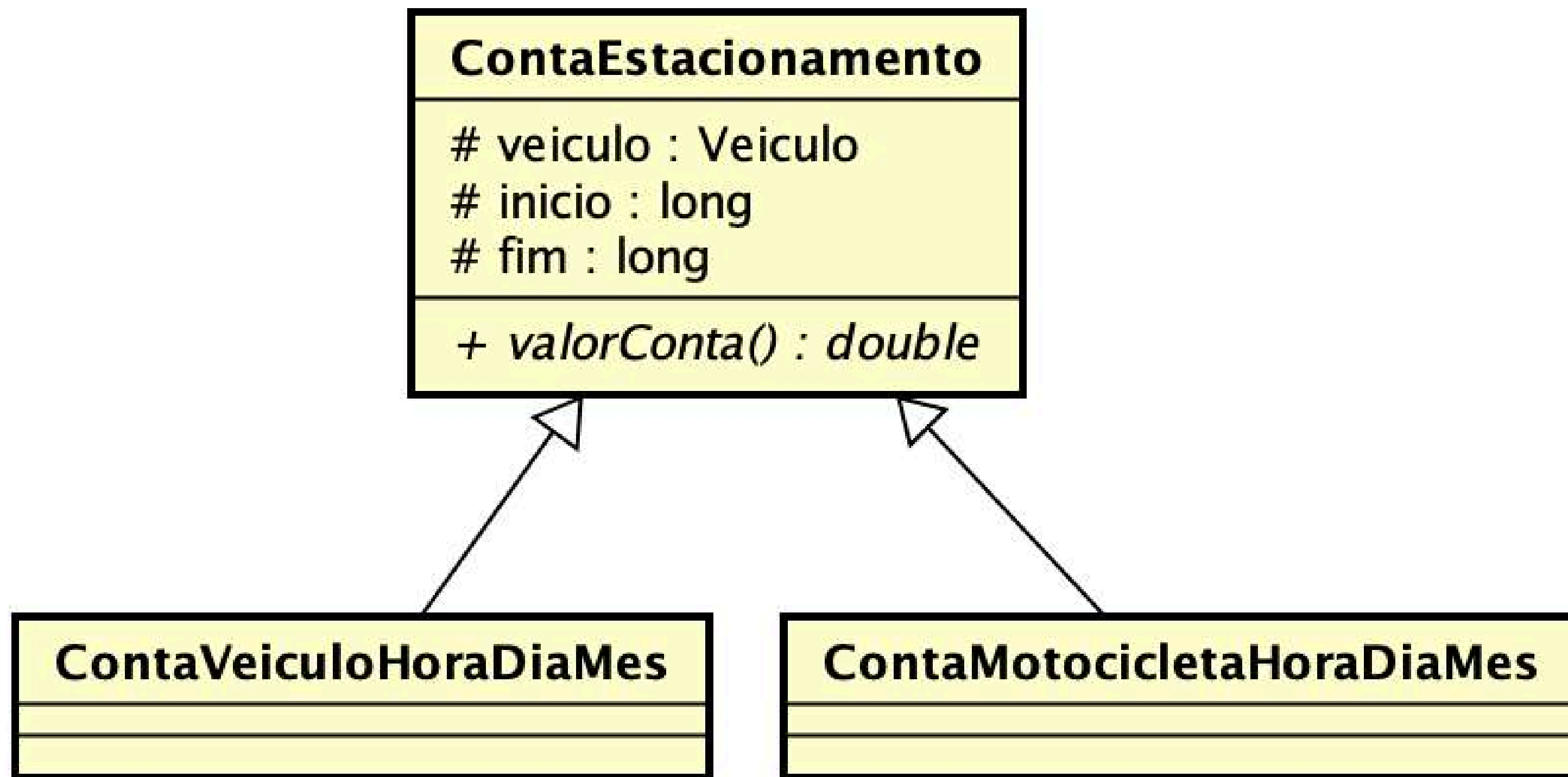
Número de classes  
Troca de tarifa/tipo de  
locação



## ***HERANÇA - SOLUÇÃO 2***

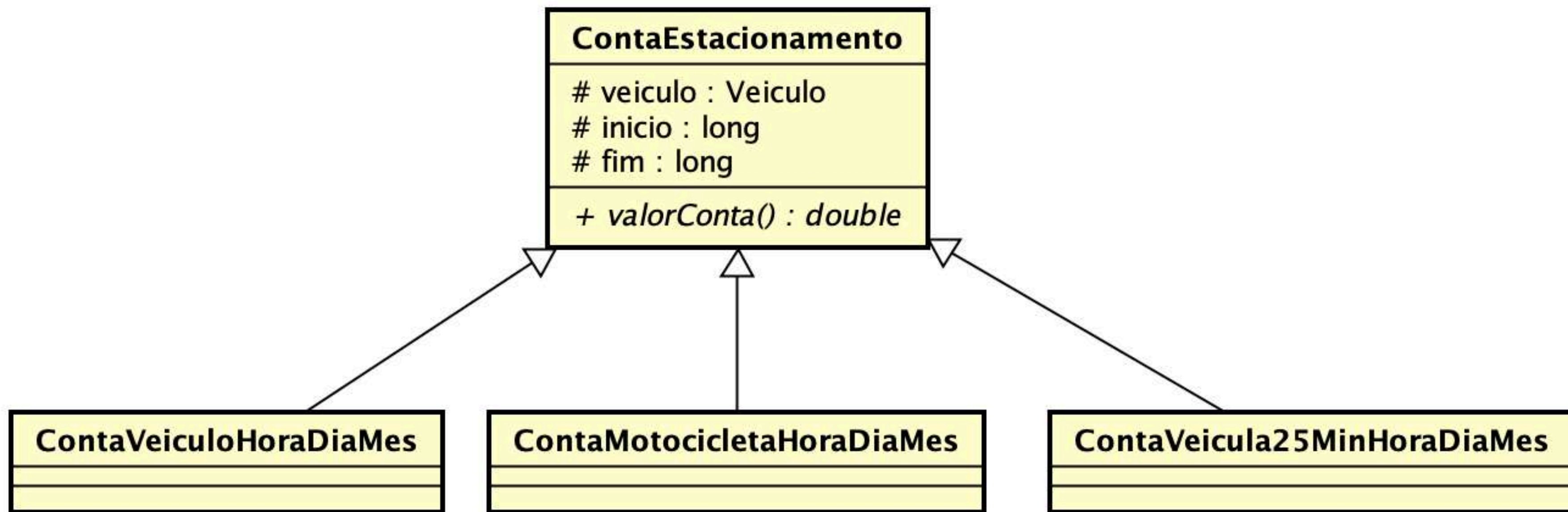
Mudar a granularidade: criar subclasses para os tipos de veículos, que implementam todas as possibilidades

# DIAGRAMA DE CLASSES DA SOLUÇÃO



# PROBLEMA

Duplicação de código. Ainda mais em pequenas variações.

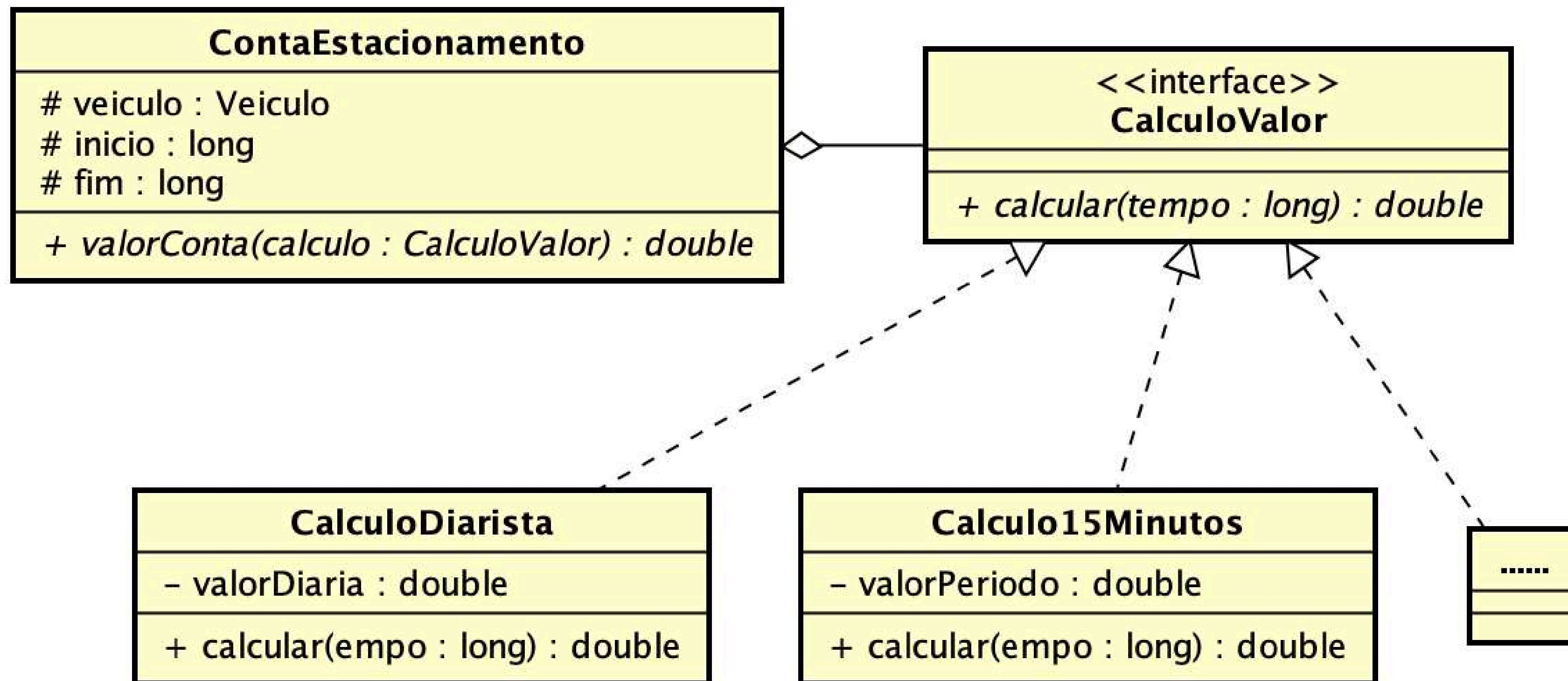




## ***QUEM SABE UMA COMPOSIÇÃO?***

Pensando em:

- Eliminar a **duplicidade**;
- Suportar a parametrização para melhor **escalabilidade**;
- Possibilidade de mudar o serviço **sem mudar a instância**.




# EM CÓDIGO...

```
public class ContaEstacionamento {
```

```
    private Veiculo veiculo;  
    private long inicio;  
    private long fim;
```

```
    public double valorConta(CalculoValor calculo) {  
        return calculo.calcular(fim - inicio);  
    }  
}
```

Abstração do  
cálculo da tarifa



# EM CÓDIGO...

```
public class CalculoDiaria implements CalculoValor {  
    private double valorDiaria;  
    public CalculoDiaria(double valorDiaria){  
        this.valorDiaria = valorDiaria;  
    }  
    public double calcular(long periodo) {  
        return valorDiaria * Math.ceil(periodo / HORA);  
    }  
}
```

Parametrização/  
escalabilidade



A large, white speech bubble with a thick black outline is positioned on the right side of the image. The background is a solid blue color with a pattern of small, light blue dots. The speech bubble has a tail pointing towards the bottom right corner.

***ESSE É UM PADRÃO***

Qual?



# ***REFERÊNCIAS***

## Referências e Material de apoio:

<https://refactoring.guru/pt-br/design-patterns/singleton>

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; Design Patters. Objected-Oriented Software. (GoF). 2004.

[https://www.dropbox.com/s/yoljem1s9288uyt/%5BMartin Fowler%2C Kent Beck%2C John Brant%2C William Opd%28Bookos.org%29.pdf?dl=0](https://www.dropbox.com/s/yoljem1s9288uyt/%5BMartin%20Fowler%2C%20Kent%20Beck%2C%20John%20Brant%2C%20William%20Opd%28Bookos.org%29.pdf?dl=0)

<https://www.dropbox.com/s/pdiqqe9lh9objgf/Design%20Patterns%20com%20java%20-%20Projeto%20orientado%20a%20objetos%20guiado%20por%20padroes%20-%20Casa%20do%20Codigo.pdf?dl=0>

<http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf>